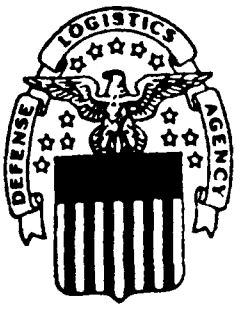
DLA-92-P20041

OPTIMAL SAMPLING PLANS FOR ITEMS REPRESENTING TWO POPULATION GROUPS

April 1992

OPERATIONS RESEARCH AND ECONOMIC ANALYSIS OFFICE



DEPARTMENT OF DEFENSE
DEFENSE LOGISTICS AGENCY

DLA-92-P20041

OPTIMAL SAMPLING PLANS FOR ITEMS REPRESENTING TWO POPULATION GROUPS

April 1992

Randal S. Wendell

DEPARTMENT OF DEFENSE
DEFENSE LOGISTICS AGENCY
OPERATIONS RESEARCH AND ECONOMIC ANALYSIS OFFICE
CAMERON STATION
ALEXANDRIA, VA 22304-6100

EXECUTIVE SYNOPSIS

The DLA Laboratory Testing Program has many objectives. The primary purpose of the random part of the program is to provide a consistent and reliable measure of the quality of DLA managed items. A secondary objective is to deter contractors from providing nonconforming material since any contract may be subjected to random testing. In an earlier study, the DLA Operations Research and Economic Analysis Office (DORO) developed the Sampling Assistance Model (SAM). SAM determines an appropriate sample size calculation and randomly selects items as candidates for testing. Based on population characteristics, confidence levels and precision levels, the sample provides valid estimates for nonconforming rates aggregated at the Defense Supply Center level. If all Centers use SAM, these estimates can be combined and reported at the DLA level.

With a fairly small sample (around 100 items), SAM can assess the overall conformance and track changes in quality at the Center level. However, data from such small samples cannot pinpoint subsets of the Centers' items that may be above or below Center goals with reasonable precision or confidence. DLA Directorate of Quality Assurance (DLA/DCMC-Q) desired a more in-depth sampling plan that would allow each Center to measure nonconformance rates by ultimate storage depot and also by contract administration activity.

DORO developed a stratified sampling plan model which simultaneously generates both sample sizes and the specific items to be sampled. The model is intended to measure, by Center, quality levels at six DLA depot complexes and at six contract administration offices (local administration plus five Defense Contract Management Districts). With this tool, each Center can reliably measure, compare and track quality rates for both depots and contract management activities. With sample sizes about one order of magnitude higher than SAM, the model provides greater resolution of quality levels.

Through the use of statistical sampling theory and advanced linear and nonlinear programming, the stratified sampling plan model accomplishes two important objectives. The plan minimizes the total sample size for each Center. This is extremely important because of the high cost to develop a test plan, pull a sample and conduct the actual laboratory test. The stratified plan also seeks to produce a sample distribution, among stratums, which mirrors the overall population distribution.

A nonlinear model using a quadratic objective function was tested along with competing alternatives and found to best meet the above objectives. This model minimizes overall sample size and the total squared difference between the sampling and population proportions of items. Thus, this model produces the best "fit" of the sample to the population. It also guarantees the lowest possible sample size to meet specified confidence and precision levels. Documentation on the mathematical formulation is provided as an attachment. The research conducted by DORO formed the basis for a Master's Thesis, successfully defended and published at Virginia Commonwealth University in May 1992.

The entire methodology was written as a user-friendly PC model. This program will enable Center Laboratory Testing personnel to enter a few variables and produce a stratified sampling plan. In addition, the program randomly selects items in accordance with the plan and produces a list of candidate items for testing. It is recommended that DLA/DCMC-Q require all DLA Hardware Supply Centers to use the stratified sampling model for the random portion of the Laboratory Testing Program. The model will provide high levels of resolution at lowest total cost to the Center. It will also allow DLA to consolidate all Supply Center data to assess quality levels from an Agency perspective.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Optimal Sampling Plans
for Items
Representing Two Population Groups

This paper examines a problem of determining optimal sampling plans. The items to be sampled belong to two distinct populations which are partitioned into sub-populations that require sampling plans. Three mathematical programming models are investigated that minimize the total sample size while also ensuring that the proportion of samples closely resemble the actual population proportions. Both linear and non-linear programming techniques are used to find an optimal sampling plan. Finally, comparisons are made from the solutions generated by "real" data for all three models.

Chapter 1 Introduction

1.1 Background

During the past several years the Federal Government has been developing new policies for purchasing supplies and equipment for the Military Services. Emphasis used to be placed on saving the taxpayer's money through purchasing from the lowest bidder (least expensive). More emphasis is now being placed on purchasing higher quality items that save more money over time. One method for ensuring the quality of an item is through laboratory testing. Testing can be expensive and time consuming, but through proper sampling, the time and money spent on testing can be kept to a minimum while yielding accurate results.

This paper examines estimating optimal sample sizes through the use of linear and non-linear programming. In the problem presented, there are two groups of distinct populations each requiring a sampling plan. The first group has M populations and the second group has N populations requiring a total of M+N sampling plans. However, the individual items to be sampled belong to two distinct populations (one from each group). Mathematical programming is applied to minimize the total sample size while ensuring the necessary sample sizes of each sampling plan.

This problem arose as a task for the Defense Logistics Agency's quality assurance program. The following paragraphs describe the prob-

lem, three possible approaches (methodologies), and comparisons of the three methodologies.

1.2 Problem Description

The Defense Logistics Agency (DLA) supplies goods that are commonly used by United States military services. Critical repair parts, food, fuel, medical supplies, and clothing must be acquired and delivered on time to the American soldiers, sailors, and airmen stationed throughout the world. DLA's goal is to support the services with quality materiel at the best value to the taxpayer.

DLA takes ma · precautions to ensure the quality of its items. However, there exists some small proportion of its items that are defective or do not meet design specification. These flawed items are referred to as nonconforming.

In an effort to improve the quality of its items, DLA initiated actions to develop a laboratory testing plan. The goals for the laboratory testing plan are to: 1) determine the percent of items that are nonconforming for all contracting offices and depots, and 2) provide this information at the lowest possible cost (fewest samples).

Currently, DLA has no sampling plan to obtain the above information and has tasked the Operations Research and Economic Analysis Management Support Office (DLA-DORO) to develop such a plan.

DLA-DORO determined that a successful sampling plan must not only provide statistically sound sample sizes, but DLA's Quality Assurance personnel must be able to easily generate a sample size in a timely manner for "what if" analysis. To meet these requirements, the sampling plan will ultimately be computed via a "user friendly" PC program. The PC program will allow various user inputs to help determine the proper sample size. Using a random number generator, the PC program will also randomly select the items for sampling.

This paper looks at three models for determining the sample sizes. The method that best meets DLA's needs will then be written as a PC program.

More information pertaining to the structure of DLA and a clearer definition of the problem is required before describing the sampling plan.

First, DLA manages many different types of items that fall into six categories (commodities). Each commodity is managed by a unique Defense Supply Center (DSC). Each DSC purchases items through one of six contract management offices. All six offices are responsible for inspecting their items (source inspection) before shipping to any one of six DLA warehouses (depots). Items are inspected again during the receiving process at the depot (destination inspection). Therefore, each DSC can inspect an item by one of thirty-six contracting office-depot combinations. In Figure 1, each arc represents a contracting office-depot combination.

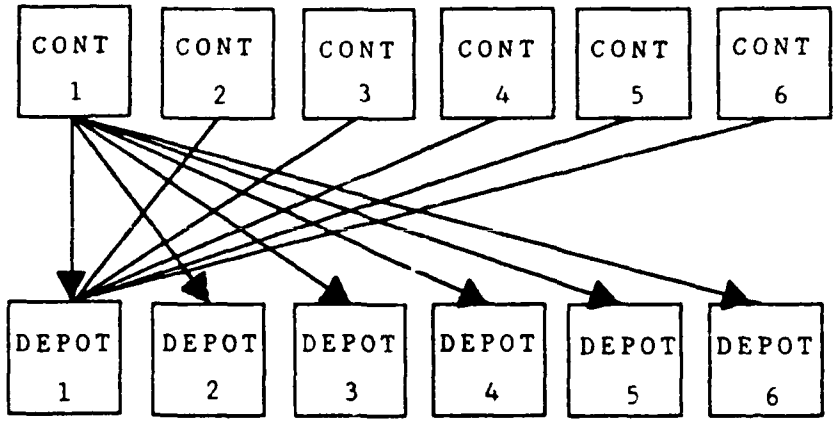


Figure 1 DSC structure

Notice that contracting office 1 has six arcs leaving while depot 1 has six arcs entering. This pattern is repeated for the remaining 5 contracting offices and depots.

Since DLA manages over 3 million different items, only a small percent of the items are actually inspected at either source or destination sites.

The goals of the sampling plan will be met by determining the nonconformance level of each contracting office and depot and to provide this information through the smallest total sample.

1.3 Components of the Sampling Model

This paper examines the generalized sampling methodology that will be used for all six DSCs. From here forward it will be assumed there are M contracting offices and N depots. Therefore, there are MN variables representing the sample sizes required for each of the M+N sampling plans.

Even though the required sample sizes represent a single DSC, the total sample size will be taken strategically to return the nonconformance percent for the M contracting offices and N depots. In other words, a single sample will represent more than one population.

Suppose the problem were to find the nonconformance percent of a single depot. The nonconformance percent could be found by taking a simple random sample from the population of items. The required size of this sample can be found from the equation below. This equation will be referred to as the simple random sampling formula and is based upon the fact that, for a simple random sample from a population, the number of nonconforming items in the sample has a binomial distribution with sample size n and nonconformance probability P. The formula is:

$$n = z^2P(1-P)/e^2,$$

where

n = the required sample size.

P = the known (historical) nonconformance level. If P is unknown then .5 is used to ensure a sample size sufficient for all possible values of P.

e = allowed error level in the estimate of the actual nonconformance level.

z = percentile for a standard normal random variable chosen to achieve the desired confidence level. The confidence level gives the probability that the resulting sample size will give the estimated true nonconformance level within the given error range.

Under most circumstances this formula will suffice. However, inaccurate results may occur when the error, e, is greater than the nonconformance level, P or (1-P). For example, suppose the nonconformance rate of a certain depot is .03 and the desired error level is $\pm$.05. The calculated sample size will be valid for the range of nonconformance rates .03 $\pm$ .05, or between -.02 and +.08. However, since the nonconformance rate can never be less than 0, the calculated sample size does not accurately reflect the chosen error level. This situation is avoided by not allowing the error level to be greater than the nonconformance rate, P, or conformance rate (1-P). Very small or very large nonconformance rates require small error levels for calculating sample size.

It is worth noting the effect each parameter has on the resulting sample size. Generally, to obtain higher confidence levels or smaller error levels costs in terms of sample size. First, the sample size increases geometrically when the allowed error is decreased. Next, as the

nonconformance level, P, approaches .5, the required sample size will increase and is maximized when P equals .5. Increasing the confidence level, thus increasing z, will also increase the required sample size.

Finding the nonconformance level of a single depot can be easily accomplished using the simple random sampling formula. The same process used for finding the nonconformance rate of a single depot could be repeated M+N-1 times for the remaining N-1 depots and M contracting offices. Together, these M+N sampling plans would yield the nonconformance level of all depots and all contracting offices and the problem would be solved. However, though this methodology provides the desired information, it will be demonstrated that the same results can be obtained using a smaller total sample size.

Recall the structure of DLA. Each DSC purchases all items through M contracting offices and then stores these items at N depots. It is possible for a single item to be used to help estimate the nonconformance level for a depot and for a contracting office. In fact, with DLA's database it is easy to identify where items are located and how they were purchased. In other words, the total sample size may be reduced by allowing items to represent both the depot and contracting office. The remaining question is how to distribute the required sample size among the MN variables and still maintain the desired confidence level for each depot and contracting office.

Mathematical programming has been chosen for solving this problem for two reasons. First, the goal, or objective of the problem, is to

minimize the total sample size. Second, this objective can only be met while satisfying the specific requirements (constraints) of the M+N sampling plans. As will be shown later, the objective function of the three models considered differ, but the same M+N sampling plan requirements (constraints) are used for each model.

Consider the sampling process which will produce the data from which nonconformance rates will be estimated. If sample size X_{ij} is selected from the population of items purchased through contracting office i and stored at depot j, and B_{ij} is the number of nonconforming items in this sample, then B_{ij} has a binomial distribution with sample size X_{ij} and nonconformance probability $P_{i.}$, where $P_{i.}$ is the proportion of nonconforming items purchased (and shipped) through contracting office i. Then, the number, $B_{i.}$, of nonconforming sample items shipped from contracting office i is

$$B_{i.} = \sum_{j=1}^{N} B_{ij}.$$

Here $B_{i.}$ is a sum of independent binomial random variables, each with nonconformance probability $P_{i.}$, so itself is binomial with sample size

$$\sum_{j=1}^{N} X_{ij} \quad \text{and nonconformance probability } P_{i.}.$$

The number, $B_{.j}$, of nonconforming sample items stored at depot j is

$$B_{.j} = \sum_{i=1}^{M} B_{ij}.$$

Here $B_{.j}$ is a sum of independent binomial random variables, each with possibly different nonconformance probabilities, since B_{ij} has nonconformance probability $P_{i.}$ In fact, the sample which will be used to estimate the nonconformance probability for depot j is a stratified random sample, each contracting office serving as a stratum. Since the number of items, N_{ij}, purchased through contracting office i and stored at depot j, is known, then the proportion of items stored at depot j which are from contracting office i,

$$N_{ij} \Big/ \sum_{i=1}^{M} N_{ij} = N_{ij}/ N_{.j} \text{ is known.}$$ The appropriate estimate for the nonconformance probability for depot j is

$$\sum_{i=1}^{M} (N_{ij}/N_{.j}) \; (B_{ij}/X_{ij}) \text{ (Cochran, 1963).}$$

Cochran further states that one working rule to use, assuming the cost of sampling an item is the same for each stratum (here, a reasonable assumption), is that the gain in precision in estimating the nonconformance probability from a stratified random sample is small or modest unless the nonconformance probabilities, $P_{i.}$, vary greatly from stratum to stratum. Indeed, the sample selected to estimate the nonconformance rate for contracting office i is a stratified random sample, with the same nonconformance rate $P_{i.}$ for each stratum. As in estimating the nonconformance rate for the depots, the sample size formula for simple random sampling can be used to provide a conservative required sample size for each contracting office.

Since the P_i do not vary greatly, the sample size formula associated with simple random sampling can be used in the planning phase of any sampling study. The sample size constraint used for depot j will be conservative, i.e., possibly larger than is actually needed to achieve the allowed error level when estimating the nonconformance rate for depot j.

Thus the simple random sample formula presented earlier is used to determine the required sample size for each depot and for each contracting office. These calculated sample sizes will be the right-hand-side values of the M+N constraints. Recall that each depot receives items bought through all M contracting offices and each contracting office buys items that are sent to all N depots. In other words, each constraint representing a particular contracting office will be the sum of N depot variables and each depot constraint will be the sum of M contracting office variables. Figure 2 displays the constraints in a two-dimensional array.

$$
\begin{array}{|cccccc|c}
X_{1,1} + & X_{1,2} + & \cdots & \cdots & + X_{1,N} & & \geq C_1 \\
X_{2,1} + & X_{2,2} + & \cdots & \cdots & + X_{2,N} & & \geq C_2 \\
\cdots & & & & \cdots & & \cdots \\
\cdots & & & & \cdots & & \cdots \\
X_{M,1} + & X_{M,2} + & \cdots & \cdots & + X_{M,N} & & \geq C_M \\
\hline
\geq D_1 & \geq D_2 & \cdots & \cdots & \geq D_N & &
\end{array}
$$

Figure 2 Contracting office-depot array and constraints

where

X_{ij} = the size of the sample taken from the population of items that were purchased through contracting office i and stored at depot j,

C_i = the number of samples required for contracting office i,

D_j = the number of samples needed for depot j.

Using sigma notation, the constraints defined in the array of Figure 2 are

$$\sum_{j=1}^{N} X_{ij} \geq C_i \text{ (for } i = 1,2,..,M)$$

$$\sum_{i=1}^{M} X_{ij} \geq D_j \text{ (for } j = 1,2,..,N)$$

$$X_{ij} \geq 0 \text{ (for } i = 1,2,...,M,\ j = 1,2,...,N).$$

These constraints possess a property that is useful in determining the best, or optimal solution, namely that the set of feasible solutions is a convex set. A set is convex if the line joining any two points on its graph lies nowhere below the graph. Fortunately, since the above constraints are all linear, they do in fact form a convex set.

The first and third models presented in this paper are solved as linear programming problems (though the third model is converted into a linear problem). An optimal solution to a linear programming problem

must lie at the point where two or more constraints intersect, i.e., at extreme points of the convex set. Because of this fact, both models are solved using the simplex algorithm.

The simplex algorithm was chosen because of its straightforward and efficient manner for finding an optimal solution. It has two major features for getting a quick result. First, the simplex algorithm will only investigate extreme points. The algorithm travels from one extreme point to the next adjacent extreme point without returning to any previous points. Next, since the constraints form a convex set, then each time an extreme point is visited the objective function value will never increase for minimization problems nor decrease for maximization problems. Extreme points that do not improve the objective value are not visited. Figure 3 will help to graphically illustrate the simplex algorithm.

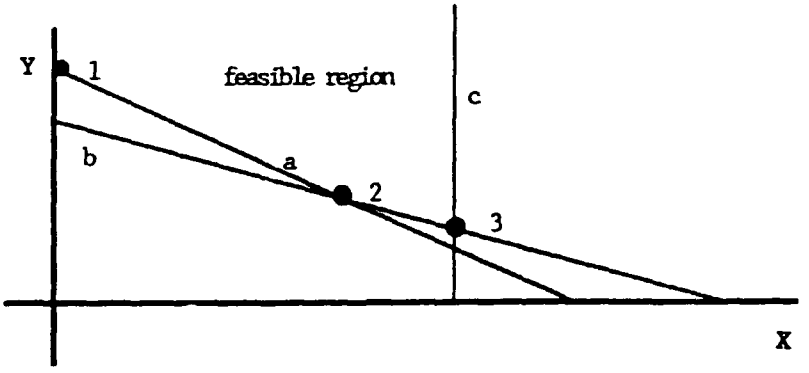


Figure 3 **The simplex algorithm**

In Figure 3, the set of feasible solutions lies in the first quadrant, above lines (a) and (b), and to the left of (c). This region

has three extreme points, (1), (2), and (3). Except for the starting point, the simplex algorithm will only investigate these extreme points, and will continue to do so until no further improvement to the objective function value is possible.

The second model is a nonlinear problem which uses a variation of the simplex algorithm. The methodology for solving this model is discussed in Chapter 3.

Another consideration for the problem is the manner in which the total sample is distributed among the depot and contracting office population for a particular DSC. Suppose a sample is taken to find the nonconformance rate of a certain depot. Since each depot contains items purchased through the N contracting offices, then more than likely, the sample will also represent each contracting office. If a disproportionate part of the sample represents a specific contracting office, then the resulting nonconformance level may reflect that contracting office and not the depot in question. Therefore, the sample distribution should reflect the contracting office-depot item population distribution of the DSC. The entire goal of the problem is to reach the minimum sample size while having the sample size be proportional to the depot-contracting office population proportion.

All three models require that their solutions (sample size) be integer. It does not make sense to laboratory test partial items. One method for obtaining integer solutions is through rounding. However, care must be taken when rounding down because this may remove the solu-

tion from the feasible region. Rounding is necessary in the second and third models, but is not needed in the first model due to its structure. This unique structure is discussed in Chapter 2.

It is useful to know both the upper and lower bounds of total sample size before actually solving the problem. An obvious lower bound for the total sample size is the maximum between the sum of required depot sample sizes and the sum of required contracting office sample sizes. The total of the required sample sizes can never be less than either the sample sizes required for all N depots or the sample sizes required for all M contracting offices. The upper bound is the sum of the total required depot sample sizes and the total required contracting office sample sizes. This sum is equivalent to the total sample sizes required if M+N individual sampling plans, as mentioned earlier, were used. If this sum is exceeded, then solving the problem as M+N individual sampling plans (without having samples representing depots and contracting offices simultaneously) as mentioned earlier, would be more efficient. Therefore, solutions of interest will yield the total sample size between

$$\text{Maximum}\left\{\sum_{i=1}^{M} C_i, \sum_{j=1}^{N} D_j\right\} \quad \text{and} \quad \sum_{i=1}^{M} C_i + \sum_{j=1}^{N} D_j.$$

Three mathematical programming approaches to finding optimal sampling plans is the focus of this paper. The objective of all three models is twofold. The first goal is to minimize the total sample size

while satisfying all constraints. The next goal is to have the proportion of samples closely resemble the actual item population proportions.

Typically, optimization problems having more than one goal are difficult to solve. In fact, many times, just defining an optimal solution may be difficult.

The models discussed in this paper in no way exhaust the possible approaches for this problem. These three models were selected for their clear interpretation, ease of solving, and reasonable solution to the underlying problem.

All three models utilize similar constraints but have different objective functions. The first model uses a weighted linear objective function. Each of the MN variables is weighted with the inverse proportion of the actual DSC population proportion. This method minimizes the total sample size while making it more costly to sample from areas of the population that make up a small portion of the true item population.

The second model applies a quadratic objective function that represents the Euclidean distance from the true DSC population proportions. Since the objective function reflects the distance between the sampling and DSC population proportions, the total sample size will grow until the minimization is met. It is therefore necessary to add an additional constraint that sets an upper bound on the sample size. To ensure the smallest possible sample size as the solution, the upper bound is set as

low as possible (usually equal to the lower bound). Due to the quadratic objective function, this model is solved by Lemke's complimentary pivoting method. A description of this method is given in Chapter 4.

The third model also minimizes the distance from the true DSC proportions, however, here the distance is defined as the maximum of all absolute deviations. Even though the objective function is not linear, the entire problem can be transformed into a linear model and solved using the simplex algorithm.

In all three models, X_{ij} denotes the sample size corresponding to contracting office i and depot j for i = 1,2,...,M, j = 1,2,...,N, and $x = [X_{ij}]$ is an MN by 1 vector.

PC programs, written in C, were used to compare the three models. Copies of the programs are provided as appendices A, B, and C. Comparisons and results are discussed in Chapter 5.

1.4 The Estimates for the Nonconformance Rates

Although the allocation of sample sizes for the contracting office-depot combinations is the main concern for this project, the method needed to estimate the nonconformance rate and the actual margin of error for each contracting office and each depot will be needed when the sampling plan is implemented. The estimator and its standard error are well known when the sample is a stratified random sample.

For contracting office i the estimator, $\Pi'_{i.}$, for the nonconformance rate, $\Pi_{i.}$, will be

$$\Pi'_{i.} = \sum_{j=1}^{N} (N_{ij}/N_{i.})(B_{ij}/X_{ij}), \quad \text{where}$$

$$N_{i.} = \sum_{j=1}^{N} N_{ij},$$

$$\text{with standard error of } \Pi'_{i.} = \sqrt{\Pi'_{i.}(1-\Pi'_{i.})/N^2_{i.} \sum_{j=1}^{N} N^2_{ij}/X_{ij}},$$

and margin of error for estimating $\Pi_{i.}$ of z(standard error of $\Pi'_{i.}$). In other words, the estimated nonconformance rate of contracting office i is the weighted sum of the nonconformance rates of each contracting office-depot combination. For depot j the estimator, $\Pi'_{.j}$, for the nonconformance rate, $\Pi_{.j}$, will be

$$\Pi'_{.j} = \sum_{i=1}^{M} (N_{ij}/N_{.j})(B_{ij}/X_{ij}), \quad \text{where}$$

$$N_{.j} = \sum_{i=1}^{M} N_{ij},$$

$$\text{with standard error of } \Pi'_{.j} = \sqrt{1/N^2_{.j} \sum_{i=1}^{M} N^2_{ij}\ \Pi'_{i.}(1-\Pi'_{i.})/X_{ij}},$$

and margin of error for estimating $\Pi_{.j}$ of z(standard error of $\Pi_{.j}$).

Chapter 2 The Linear Model

2.1 Linear Model Formulation

The first model is a linear programming model in which the two goals are incorporated into the objective function. Recall that the task is to minimize the total sample size while fitting the actual item population proportion. Therefore, the objective equation is weighted to reflect the item distribution for the DSC. Due to the objective equation being a minimization, the weights will be the inverse proportion of the actual population proportion. Thus the objective function is a weighted sum of sample sizes.

$$\text{Minimize } f(x) = \sum_{i=1}^{M} \sum_{j=1}^{N} w_{ij} X_{ij}$$

where

X_{ij} = the size of the sample taken from the population of items purchased through contracting office i and stored at depot j,

x = $[X_{ij}]$ is an MN by 1 vector,

w_{ij} = a weighting factor that equals $1/(p_{ij}+1)$ with p_{ij} being the proportion of items purchased through contracting office i and stored at depot j.

The above objective function is subject to the M+N constraints presented in section 1.2.

Another desire, though not a requirement of the model, is to not allow any X_{ij}s to equal zero. Even though X_{ij} may be too small to be significant, there may exist enough information to warrant further investigation. Letting X_{ij} equal zero, however, will not provide any information for that portion of the population. For this reason, a fixed lower bound, β_{ij}, is set for each variable X_{ij}.

The lower bound β_{ij} is found by first determining the maximum between the sum of required depot sample sizes and the sum of required contracting office sample sizes (as discussed in section 1.3), called L^*. Notice that to have identical proportions means that $X_{ij} = L^* p_{ij}$ for each i and j, where p_{ij} is the actual proportion of items purchased through contracting office i and stored at depot j. However, this exact fit comes at the cost of a large sample size. Therefore, $\beta_{ij} = L^*(p_{ij} - \tau_{ij})$ where τ_{ij} is some fixed tolerance level (a percent of p_{ij}) and $L^*(p_{ij} - \tau_{ij}) \geq 0$. The Linear Model, in its entirety, is formulated below:

$$\text{Minimize } f(x) = \sum_{i=1}^{M} \sum_{j=1}^{N} w_{ij} X_{ij},$$

subject to:

$$\sum_{j=1}^{N} X_{ij} \geq C_i \quad (\text{for } i = 1,2,\ldots,M)$$

$$\sum_{i=1}^{M} X_{ij} \geq D_j \quad (\text{for } j = 1,2,\ldots,N)$$

$X_{ij} \geq \beta_{ij}$ (for i = 1,2,...,M, j = 1,2,...,N)

and X_{ij} is integer.

2.2 Linear Model Solution Procedure

The Linear Model is an integer programming problem. In general, integer optimization problems, even those with linear constraints and linear objective functions, are difficult to solve and require special techniques for obtaining a solution. Fortunately, due to the structure of the formulated Linear Model, using the simplex algorithm will produce an all integer solution. The reason for this lies in the following paragraphs.

Using matrix notation, the constraints can be expressed as Ax = b. Matrix A can be partitioned as (B,G), and similarly, vector x can be partitioned as (x_B, x_G).

If B^{-1} exists, then it is useful to express the constraints as $Bx_B + Gx_G = b$, where x_B is a vector of basic variables and x_G is a vector of nonbasic variables. In this case, $x_G = 0$, and solving for x_B yields $x_B = B^{-1}b$. Note that this equation is equivalent to $x_B = adj(B)\ b\ /\ det(B)$ where adj(B) is the adjoint of B. Since adj(B) is the transpose of signed minors of B, its entries will always be integer. By definition of the problem, entries of vector b are also integer numbers. Therefore, one way to guarantee that each basic variable in x_B be an integer, is for det(B) to be either +1 or -1.

The next several paragraphs prove that if B is a square submatrix of A, then det(B) is indeed 0, +1, or -1.

Matrix A is said to be unimodular if the determinant of every square submatrix of A is either 0, +1, or -1. In other words, A is unimodular if and only if every submatrix of A is unimodular.

To show unimodularity, first let the combined coefficients of the constraints and lower bounds make up a (M+N + MN) by (MN) matrix A. The constraint coefficients make up a (M+N) by (MN) submatrix A' while the lower bounds create a (MN) by (MN) identity matrix . Therefore

$$A = \begin{bmatrix} A' & -I & 0 \\ \\ I & 0 & -I \end{bmatrix}$$

From the above matrix, it can be seen that each element of matrix A is either 0, +1, or -1. Therefore, every square submatrix of size 1 has determinant of either 0, +1, or -1.

Each column of matrix A may contain up to three 1's (both +1 and -1). Matrix A' contains two 1's, matrix I contains one 1 in every column, and matrix -I contains one -1 in every column.

To show that matrix A is unimodular, four different cases must be investigated and proved through induction. First, suppose the determi-

nant of all submatrices of A of size up to (m-1) is either 0, +1, or, -1 and let B be a square submatrix of size m.

Case 1. Let B be a square submatrix of matrix A containing a column with no 1's. By definition det(B) = 0 and B is unimodular.

Case 2. Let B contain a column with one 1 (either +1 or -1). Now, expanding on the column containing the single 1 we have det(B) = $\pm$ det(B') where B' is the submatrix of B resulting from the deletion of the column containing the single 1 and the corresponding row. From the hypothesis, det(B') = 0, +1, or -1, which implies that det(B) = 0, +1, or -1, and B is unimodular.

Case 3. Let every column of B contain at least two 1's (else see case 2).

Case 3a. All rows belonging to matrix A' represent the depot and contracting constraints. By definition, the sum of the depot rows equals the sum of the contracting rows, which is a vector of 1's. Therefore, submatrix B has a rank (the number of linearly independent rows) less than m. In other words, det(B) = 0.

Case 3b. At least one row belongs to unit matrix I or -I. By expanding on the row contained in I or -I we have the same situation that appeared in case 2. All remaining elements of that row are zero and B is unimodular.

Case 4. Let every column of B contain three 1's. Since at least one row is contained in I, B is unimodular as in case 3b.

Thus each extreme point of the feasible region is guaranteed to be integer and the simplex method can be applied to find an optimal solution to the Linear Model.

Chapter 3 The Quadratic Model

3.1 Quadratic Model Formulation

The second model uses a quadratic objective function with linear constraints. Here the total squared distance between the sampling population proportion and the actual DSC item population proportion is minimized. The objective function of the Quadratic Model, to be minimized, is:

$$f(x) = \sum_{i=1}^{M} \sum_{j=1}^{N} (X_{ij} - p_{ij}L)^2$$

where

L = a chosen upper bound for the total sample size,

p_{ij} = the proportion of total items purchased through contracting office i stored at depot j for a given DSC.

Figure 4 is a graphical representation of the Quadratic Model's objective function. The concentric circles represent the contours of the quadratic objective function. All points on each circle are at the same Euclidean distance from its center. The closer the solution is to the center of the circle the better the fit to the actual population. If the solution is allowed to equal the center of the circle, then the fit is perfect and the difference between sampling population proportion and the DSC item population proportion is zero.

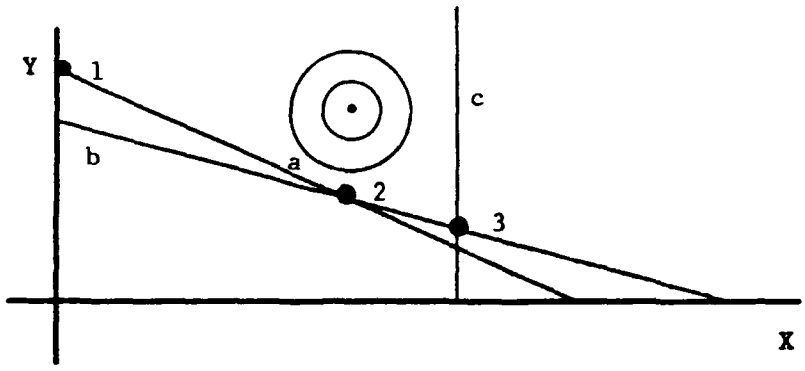


Figure 4 **The quadratic objective function**

As in the Linear Model, the first M+N constraints are required to obtain the nonconformance level for each depot and contracting office. However, since the Quadratic Model's objective function is designed so that each X_{ij} approaches some proportion of the upper bound, the M+N lower bound constraints are no longer required. Instead, a new single constraint, limiting the sum of X_{ij} to the upper bound, L, is added to the problem. Usually L will be chosen as small as possible

$$\text{(e.g., set } L = \text{maximum}\Big(\sum_{i=1}^{M} C_i,\ \sum_{j=1}^{N} D_j\Big)\text{)}$$

The entire Quadratic Model is as follows:

$$\text{Minimize } f(x) = \sum_{i=1}^{M}\sum_{j=1}^{N} (X_{ij} - p_{ij}L)^2$$

subject to:

$$\sum_{j=1}^{N} X_{ij} \geq C_i \text{ (for } i = 1,2,..,M)$$

$$\sum_{i=1}^{M} X_{ij} \geq D_j \text{ (for } j = 1,2,\ldots,N)$$

$$\sum_{i=1}^{M} \sum_{j=1}^{N} X_{ij} \leq L$$

$$X_{ij} \geq 0 \text{ (for } i = 1,2,\ldots,M,\ j = 1,2,\ldots,N).$$

Recall that minimizing the total sample size is still of major concern. Without the third constraint the total sample size will increase until the perfect fit is found. From a practical standpoint, a "perfect" fit is usually not needed but a "good" fit is necessary. The third constraint can be used for determining the proper balance between sample size and a sufficient fit.

It may be of interest to know how large is the total sample size required for a perfect fit The next section includes the examination of this number.

3.2 Relevant Properties of the Quadratic Model

The first property of the Quadratic Model is that as the upper bound L increases, the total squared deviations approach 0. To calculate the upper bound that achieves this perfect fit, the contracting office or depot requiring the most samples must first be identified. If, by using a large enough upper bound, a perfect fit is met for this depot or contracting office, then the remaining depots and contracting offices will

have a perfect fit as well. First note that the squared deviations equal 0 when $X_{ij} = p_{ij}L$. For the solution to be feasible, L must satisfy the following inequalities:

$$\sum_{j=1}^{N} X_{ij} - \sum_{j=1}^{N} p_{ij} L \geq C_i \quad (\text{for } i = 1,\ldots,M),$$

$$\sum_{i=1}^{M} X_{ij} - \sum_{i=1}^{M} p_{ij} L \geq D_j \quad (\text{for } j = 1,\ldots,N).$$

Therefore, if

$$L \geq \max_{i,j}\left(C_i \Big/ \sum_{j=1}^{N} p_{ij},\ D_j \Big/ \sum_{i=1}^{M} p_{ij} \right),$$

then all variables will have a perfect fit.

Usually, solving a nonlinear programming problem is considerably more difficult than solving a linear programming problem. However, being quadratic, the second model has properties that allow for a relatively easy solution.

Suppose the problem were to minimize some nonlinear differentiable objective function, h(x), subject to no constraints (unconstrained). An optimal solution would occur at a vector x^* where the first partial derivative $\nabla h(x^*) = 0$. Having $\nabla h(x^*) = 0$ is a necessary condition but it does not guarantee that x^* is an optimal solution (only that x^* may be one of many potential optimal solutions). However, if h(x) were a convex function, then this would be a sufficient condition. And, if h(x)

were a strictly convex function , then there could exist only one x^*, where $\nabla h(x^*) = 0$, and this vector would be the unique optimal solution. One method for finding x^*, is to solve the system of simultaneous equations that set all partial derivatives $\nabla h(x^*)$ to zero. In case of constrained optimization, necessary and sufficient conditions for a vector x^* to be optimal are referred to as Karush-Kuhn-Tucker conditions.

These conditions will be prescribed for the following problem with linear constraints, called problem A:

minimize $h(x)$

subject to

$$Ax \geq b$$

$$x \geq 0.$$

The Karush-Kuhn-Tucker conditions state that if $h(x)$ is a convex quadratic function, that is

$$h(x) = .5x^tHx + c^tx,$$

where $H(x)$ is a positive semidefinite matrix, then x^* is an optimal solution if and only if there exist vectors y, u, and v which solve the system

(1) $-Hx^* - A^t u + v = c$

(2) $-Ax^* + b + y = 0$

(3) $u^t[-Ax^* + b] = 0$

(4) $x^{*t}v = 0$

$x^*, y, u, v \geq 0.$

Moreover, if H(x) is a positive definite matrix, then h(x) is strictly convex, in which case an optimal solution is unique.

Notice from equation (2) that $y = Ax - b$. Substituting y into equation (4) produces $u^t y = 0$. The system of Karush-Kuhn-Tucker equations is now

(1) $-Hx - A^t u + v = c$

(2) $-Ax + b + y = 0$

(3) $u^t v = 0$

(4) $x^t v = 0$

$x, y, u, v \geq 0.$

The constraints $x^t v = 0$ and $u^t y = 0$ have special meaning in that they do not allow for both x and v, or u and y, to be basic variables at the same time when considering feasible solutions. Pairs of variables where only one is allowed in the basis at a time are called complementary variables.

Using matrix notation, the objective function can be rewritten as $f(x) = x^t x - 2Lp^t x + p^t p$. This is a quadratic function, and this func-

tion is strictly convex. Strict convexity can be easily verified by noticing that the Hessian matrix H of f(x) is positive definite. The Hessian matrix of f(x) is the matrix of the second partial derivatives of f(x). For the case of f(x) in the Quadratic Model, H = 2I. Thus the quadratic form $x^tHx = 2x^tx$ and is positive for all nonzero vectors x.

With this information, the Karush-Kuhn-Tucker conditions can be utilized in determining an optimal solution.

3.3 Quadratic Model Solution Procedure

Expressing the Karush-Kuhn-Tucker conditions as a linear complementary problem allows use of Lemke's complementary pivoting algorithm.

The linear complementary problem is to find vectors w and z such that

$$w - Mz = q$$

$$w, z \geq 0$$

$$w^tz = 0.$$

If q is not ≥ 0, as is the case of the Quadratic Model, then a new column 1 and an artificial variable are introduced, resulting in the following system.

$$w - Mz - 1z_o = q$$

$$w, z, z_o \geq 0$$

$$w^tz = 0.$$

The initial basic feasible solution is found by letting z_o = maximum$\{-q: q_i \leq 0\}$, $z = 0$, and $w = q + 1z_o$. Lemke's algorithm makes pivots until the artificial variable, z_o, is driven out of the basis. The algorithm also excludes an entering basic variable whose complementary variable is currently a basic variable. The choice for the entering variables is made among the remaining nonbasic variables by the minimum ratio test used in the simplex algorithm. Rows and columns are updated as in the simplex algorithm.

In order to apply Lemke's algorithm to solve the Karush-Kuhn-Tucker system, matrix M and variables q, w, and z need to be identified.

Note that equations (1) and (2) may be expressed as

$$0 - Ax + y = -b$$

$$-A^t u - Hx + v = c.$$

The complementary variables are represented by the manner in which Lemke's algorithm selects entering variables.

Now letting

$$M = \begin{bmatrix} 0 & -A \\ A^t & H \end{bmatrix}, \quad q = \begin{bmatrix} -b \\ c \end{bmatrix}, \quad w = \begin{bmatrix} y \\ v \end{bmatrix}, \quad z = \begin{bmatrix} u \\ x \end{bmatrix},$$

where c = 2LP, allows Lemke's algorithm to be applied to the Quadratic Model.

One minor drawback to the Quadratic Model is that unimodularity is lost and integer solutions are no longer guaranteed. However, carefully rounding the solution (not violating any constraints) will maintain feasibility.

Chapter 4 The Maximum Deviation Model

4.1 Maximum Deviation Model Formulation

The third model minimizes the distance from the true DSC population proportion, however, distance is now defined as the maximum absolute value of the difference between the sampling population and the actual population.

The Maximum Deviation Model's objective function is not linear, but can be transformed into a linear function by adding several variables. Figure 5 is a graphical representation of the third model.

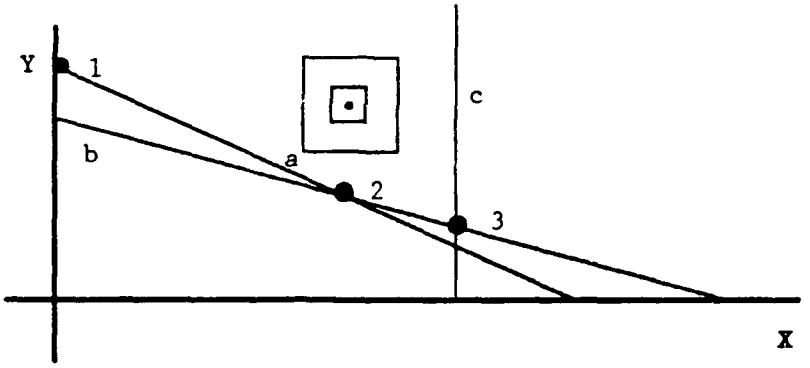


Figure 5 **Minimizing the maximum absolute deviation**

Again, the center point is where the perfect fit lies. Due to the absolute value function, the contours of the objective equation form squares. All points on a given square are the same distance from the perfect fit.

The Maximum Deviation Model is currently expressed as:

$$\text{Minimize } z = \text{Maximum}_{ij} \left| X_{ij} - p_{ij}L \right|$$

subject to

$$\sum_{j=1}^{N} X_{ij} \geq C_i \quad (\text{for } i = 1,2,..,M)$$

$$\sum_{i=1}^{M} X_{ij} \geq D_j \quad (\text{for } j = 1,2,..,N)$$

$$\sum_{i=1}^{M} \sum_{j=1}^{N} X_{ij} \leq L$$

$$X_{ij} \geq 0 \ (\text{for } i = 1,2,\ldots,M,\ j = 1,2,\ldots,N),$$

where

$|\alpha|$ = the absolute value of any argument α and L is the upper bound described in Chapter 3.

The first difficulty arising from this type of objective function is in expressing the absolute value as a linear function.

Notice that the difference $X_{ij} - p_{ij}L$ may be positive or negative, but by taking the absolute value of this difference, a nonnegative number will always result. One way to capture this expression linearly is to let

$$X_{ij}^{+} - X_{ij}^{-} = X_{ij} - p_{ij}L$$

where both X_{ij}^+ and X_{ij}^- are ≥ 0. The equivalent linear objective function is

$$z = \underset{ij}{\text{Maximum}} \, (X_{ij}^+ + X_{ij}^-).$$

Therefore, the following constraints must be added for each i and j:

$$X_{ij} - X_{ij}^+ + X_{ij}^- = p_{ij}L$$

$$X_{ij}^- , X_{ij}^+ \geq 0.$$

Now to express the current objective function as a single minimization problem let $z \geq X_{ij}^+ + X_{ij}^-$ and add the following constraints:

$$z - X_{ij}^+ - X_{ij}^- \geq 0, \; i=1,\ldots,M, \; j=1,\ldots,N.$$

The objective function is now

Minimize z.

Notice that $z \geq X_{ij}^+ + X_{ij}^-$ for each i and j. These equations, combined with the objective function, imply that the largest $X_{ij}^+ + X_{ij}^-$ equals the smallest z. Therefore, an optimal solution will make the largest $X_{ij}^+ + X_{ij}^-$ as small as possible. Thus the Maximum Deviation Model has been fully converted into a linear programming program that may be

solved using the simplex algorithm. The Maximum Deviation Model, in its entirety is shown below.

Minimize z

subject to

$$z \geq X_{ij}^{+} + X_{ij}^{-} \qquad \text{(for } i = 1,2,\ldots,M,\ j = 1,2,\ldots,N\text{)}$$

$$X_{ij} + X_{ij}^{+} - X_{ij}^{-} = p_{ij}L \qquad \text{(for } i = 1,2,\ldots,M,\ j = 1,2,\ldots,N\text{)}$$

$$\sum_{j=1}^{N} X_{ij} \geq C_i \qquad (i=1,\ldots,M)$$

$$\sum_{i=1}^{M} X_{ij} \geq D_j \qquad (j=1,\ldots,N)$$

$$X_{ij},\ X^{+}_{ij},\ X^{-}_{ij} \geq 0 \text{ (for } i = 1,2,\ldots,M,\ j = 1,2,\ldots,N).$$

4.2 Maximum Deviation Model Solution Procedure

The Maximum Deviation Model has been transformed to a linear programming problem which can be solved by the simplex method. However, due to the added constraints, unimodularity has been lost and the solution may not be integer. Again, as in the Quadratic Model, the solution can be rounded and still maintain its feasibility.

Chapter 5 Implementation

5.1 Matrix Reduction Techniques

Ultimately, the model which best meets DLA's needs will be written as a user friendly PC program. Because DLA has six contracting offices and six depots, resulting in thirty-six variables, none of the models appear too large for PC use. However, some of the models require additional variables and constraints and some require surplus and artificial variables for conversion to simplex form.

Recall that the Linear Model requires thirty-six additional constraints for its lower bounds. These new constraints also require thirty-six surplus variables and thirty-six artificial variables. After converting the Linear Model into standard simplex form, a forty-eight by one-hundred-and-thirty-three matrix was required to solve the problem. Aside from taking up crucial PC memory, the model also ran slowly due to the number of calculations required for the numerous rows and columns.

Two matrix reduction techniques were employed to speed up model execution time. In the Linear Model, the thirty-six lower bound constraints, $X_{ij} \geq \beta_{ij}$, were eliminated by a simple transformation. Letting $X'_{ij} = X_{ij} - \beta_{ij} \geq 0$, allows substituting X'_{ij} for $X_{ij} + B_{ij}$ and thus removing all thirty-six lower bound equations. After solving the Linear Model, the transformation is reversed by solving for X_{ij}.

Another technique was used to reduce the number of artificial variables needed to represent the first twelve constraints in standard form.

First, all twelve constraints were multiplied by -1 to convert the constraints to "$\leq$" equations as opposed to their original "$\geq$" signs. The constraints were then converted into equalities by adding slack variables. Note that at this time the right-hand-side of all twelve constraints is negative, which is not allowed in the simplex algorithm. Next, the equation that had the most negative right-hand-side value was subtracted from each of the remaining eleven constraints. Now these eleven constraints have non-negative right-hand-side values and the simplex algorithm can be applied. All that remains is to multiply the remaining constraint by -1 and add an artificial variable.

These techniques are mentioned only because they proved helpful in the execution time for all three models. They proved particularly useful for the Maximum Deviation Model, which with its many additional variables, was originally too large for PC programming.

5.2 Results

Two performance indicators were used to compare the three models. The first indicator was easy to measure since it examined the total required sample size. Obviously, smaller sample sizes are more desirable than larger sample sizes. The second indicator is made up of three comparisons, each designed to measure how closely the distribution of sample

sizes fits the DSC population. The three comparisons are by: 1) total absolute proportion deviation 2) total squared proportion deviation, and 3) maximum proportion deviation.

These comparisons were made using as much real data as possible. The actual population proportions were obtained from DLA's database and all confidence and error levels were set to values which would typically be used by DLA (95% and 5%, respectively). Nonconformance rates were estimates using prior information.

All three models performed equally well for minimizing the total sample size of this example. Since the sum of required depot sample sizes was 572 and the sum of the required contracting office sample sizes was 611, the lower bound in all three problems was 611. All three models produced 611 as the minimum sample size while distributing the samples differently. However, the Quadratic Model and the Maximum Deviation Model, as expected, produced non-integer solutions that required rounding. As a result, their sample sizes rounded to numbers slightly higher than 611. See Figure 6 for these results.

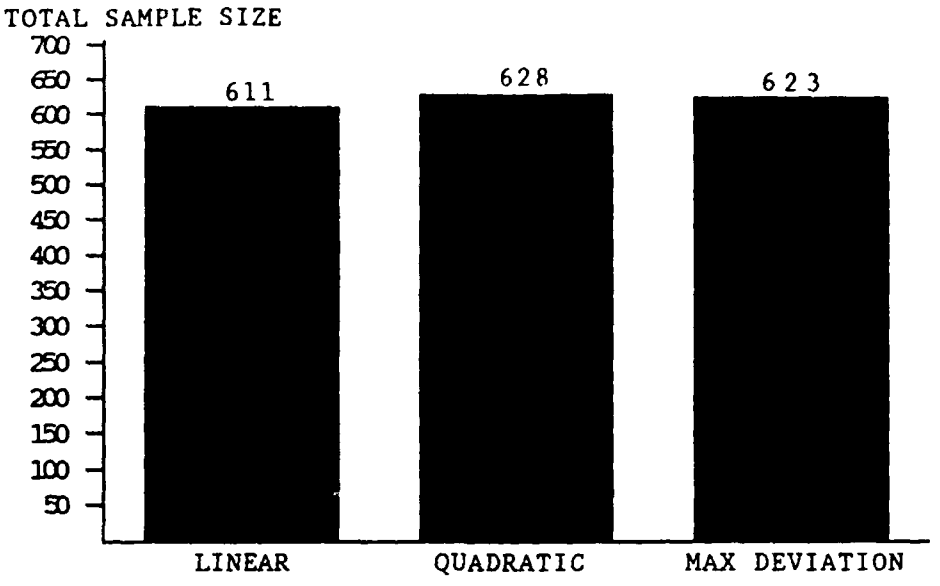


Figure 6 **Minimum sample size**

As seen in Figure 6, there is no noticeable difference between the three models for obtaining the minimum required sample size.

The next three graphs compare the models by how well their solutions fit the DSC item population proportion. These comparisons not only distinguish fit at the minimal sample size, but also investigate fit for sample sizes that exceed the possible lower bound. Even though the minimal sample size is 611, in this example, the upper bound is increased in search of a superior fit. This may be of interest for determining the cost of obtaining a better fit. The first of these graphs, Figure 7, examines the total absolute proportion difference of each model's solution.

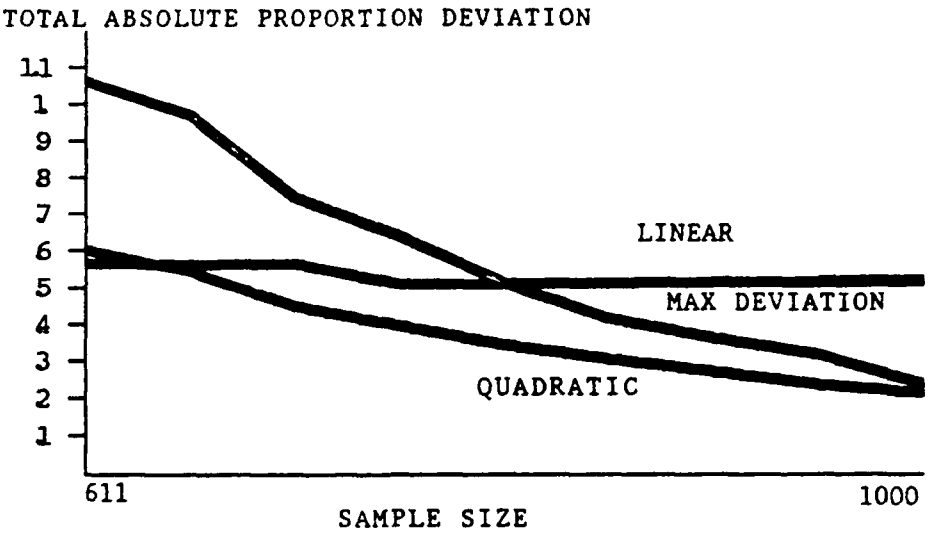


Figure 7 **Total absolute proportion deviation**

Notice that the linear model showed no improvement as the sample size (lower bound) increased. This results from the Linear Model's tolerance level being constant. As the sample size is increased, the tolerance level may be reduced giving a closer fit to the actual population. It is also important to notice that the Maximum Deviation Model did not perform well for small sample sizes. Recall that this model minimizes the maximum absolute deviation and does not minimize the total absolute deviation.

Figure 8 depicts the total squared proportion deviation of the three model's solutions.

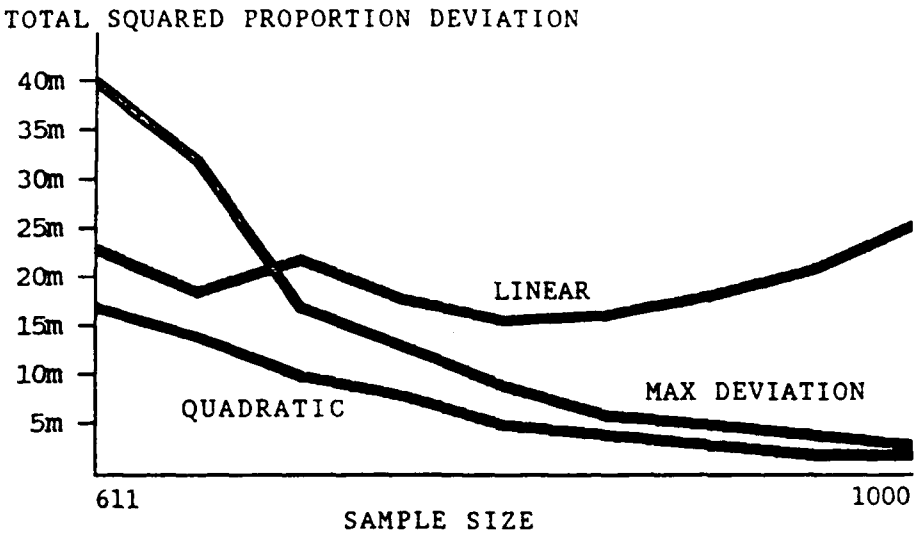


Figure 8 Total squared proportion deviation

As expected, the Quadratic Model, whose objective function parallels this comparison criterion, performed better than the other two models. Again, the constant tolerance level of the Linear Model kept it from performing as well with large sample sizes.

In Figure 9 is shown a comparison of the models according to the maximum deviation of all variables criterion.

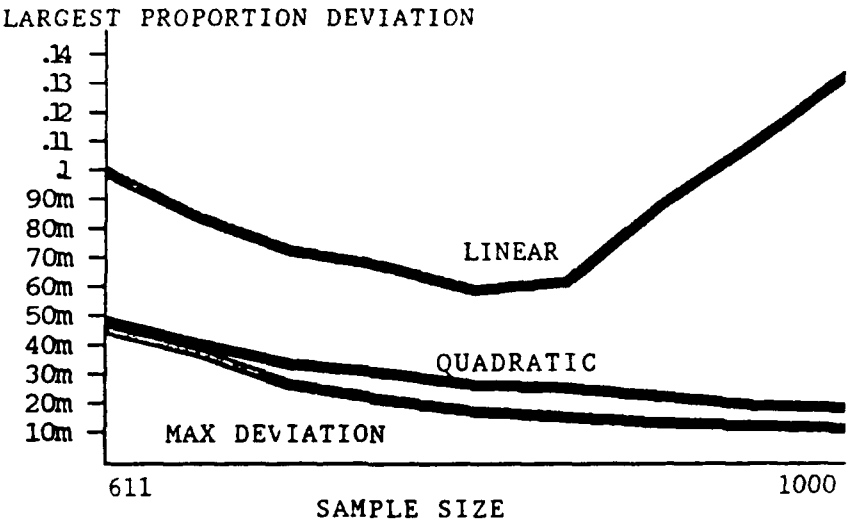


Figure 9 **Maximum proportion deviation**

This criterion paralleled the Maximum Deviation Model's objective function, resulting in its good performance. It is interesting to note that for the Linear Model, there was actually increased maximum deviations as the sample size increased. This is due to the fact that the Linear Model was designed to perform well using the minimum upper bound. Recall that the individual lower bounds are some fraction of the overall lower bound (which in this case is 611). The tolerance level is set fairly "loose" to make the overall lower bound attainable. A "tighter" tolerance level is beneficial for larger sample sizes, but prevents the model from reaching the minimal sample size.

Another area for consideration, specific to DLA, is PC execution time. Many of DLA's computers lack math co-processors causing programs with intensive mathematical operations to run slowly.

The Maximum Deviation Model has considerably more variables than the other two models and requires many more computations for a solution. Figure 10 shows the typical PC clock time for the three models. This test was run on a 33MHz PC that did not have a math co-processor.

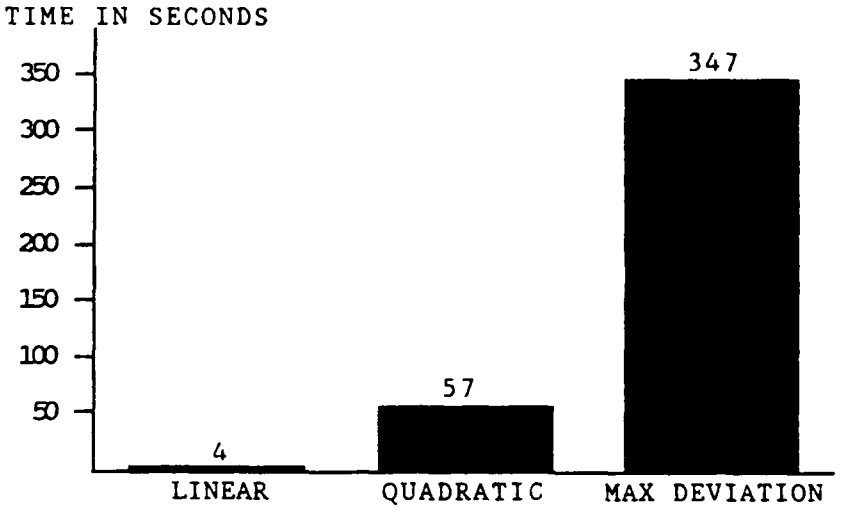


Figure 10 PC execution time

Arguments can be made for the effectiveness of all three approaches, depending on what is important to the decision maker. In this example, the Quadratic Model was the overall superior performer.

Bibliography

Basaraa, Mokhtar S., and Shetty, C.M., *Nonlinear Programming Theory and Algorithms*, John Wiley & Sons, New York, NY, 1979.

Cochran, William G., *Sampling Techniques*, second edition, John Wiley & Sons, New York, NY, 1963.

Lee, Sang M., Moore, Laurence J., and Taylor, Bernard W. III *Management Science*, second edition, WCB, Dubuque, IA, 1985.

Luenberger, David G., *Linear and Nonlinear Programming*, second edition, Addison-Wesley, Reading, MA, 1984.

Hillier, Frederick S., and Lieberman, Gerald J., *Introduction to Operations Research*, third edition, Holden-Day, Oakland, CA, 1980.

Salkin, Harvey M., and Mathur, Kamlesh, *Foundations of Integer Programming*, North-Holland, New York, NY, 1989.

Walpole, R., and Myers, K, *Probability and Statistics for Engineering and the Scientists*, third edition, MacMillan, New York, NY, 1985.

Yamane, T., *Elementary Sampling Theory*, Prentice-Hall, Englewood Cliffs, NJ, 1967.

Appendix A

C code used for solving the Linear Model

```
/*                        MODEL 1                                    */
/*  Model 1 is solved using the simplex algorithm                    */

#include <conio.h>
#include <alloc.h>
#include <stdio.h>

/*  void message(int); */

struct basis_info
{
      float cj_value;
      int cj_index;
} BASIS[60];

void main()
{

      float a[13][51];
      float cj[60];
      float zj[60];
      float samp[37];
      float p[37];
      float pivot = 0;
      float max = 0;
      float min = 0;
      float row_factor = 0;
      float minimum_ratio = 0;
      float z_calc = 0;
      float total_samples = 0;
      float l_bound = 14;

      int enter, leave, b, iterations, exx, why;
      int i, j, optimal, b_vector, unbounded, columns, rows;
      int err_message, b_var;
      int test1;

      clrscr();
      textbackground(BLUE);
      textcolor(WHITE);
      columns = 50;
      rows = 12;
      b_vector = columns;
      iterations = 0;
      err_message = 0;
```

```
/* insert the actual DSC proportions here                          */


/*  inititialize all coefficients to zero                          */
for (i=1; i<=rows; i++)
{
      samp[i] = 0;
      for (j=1; j<=columns; j++)
      {
            a[i][j] = 0.0;
      }
}

/*       set up constraint column coefficients to equal -1        */
for (i=1; i<=6; i++)
{
      for (j=1; j<=6; j++)
      {
            a[i][i+(j-1)*6] = -1.0;
      }
}


/*    set up constraint row coefficients to equal -1              */
for  (i=7; i<=12; i++)
{
      for (j=1; j<=6; j++)
      {
            a[i][(i-7)*6+j] = -1.0;
      }
}

/*    set up constraint slack variables                           */
for  (i=1; i<=12; i++)
{
      a[i][36+i] = 1.0;
}/*       column and row RHS values                               */
/*  lower bounds incorporated into constraints                   */
/* insert RHS of first 12 constraints here                       */


max = 0;
/*  find constraint with most negative RHS                       */
for (i=1; i<=12; i++)
{
  if (a[i][b_vector] < max)
  {
        enter = i;
        max = a[i][b_vector];
  }
}
```

```
/*  subtract most negative constraint from remaining constraints */
   for (i=1; i<=12; i++)
   {
         if (i != enter)
         {
               for (j=1; j<=columns; j++)
               {
                     a[i][j] = a[i][j] - a[enter][j];
               }
         }
   }

  /*  mulitply the max row by -1                                    */
  for (j=1; j<=columns; j++) a[enter][j] = -a[enter][j];

  /*  add the single artificial variable                            */
  a[enter][49] = 1.0;

   /*  the objective function coefficients                          */
   for (i=1; i<=36; i++) cj[i] = 1/(p[i]+1);

   /*  the slack and surplus variable's coefficients                */
   for (i=37; i<=48; i++) cj[i] = 0.0;

   /*    artificial variable-big Mts                                */
   cj[49] = 1000000;

   /*  set the slack variableless and artificial as basic           */
   for (i=1; i<=12; i++)
   {
         if (i != enter)
         {
               BASIS[i].cj_index = i+36;
               BASIS[i].cj_value = cj[i+36];
         }
         else
         {
               BASIS[i].cj_index = 49;
               BASIS[i].cj_value = cj[49];
         }
   }

   optimal = 1;
   clrscr();
   gotoxy(10,5);
   cputs("ITERATIONS # ");
```

```
/*  the simplex algorithm                                          */
while ((optimal == 1) && (err_message == 0))
{
       gotoxy(30,5);
       cprintf(" %d ",iterations);
iterations = iterations + 1;

      max = 0;
      optimal = 0;

      for (j=1; j<=columns-1; j++)
      {
        z_calc = 0;
        for (i=1; i<=rows; i++)            /*  calculate Zjs  */
        {
            {
            z_calc = z_calc + a[i][j]*BASIS[i].cj_value;
                  }
        }
        zj[j] =  z_calc;
        if (zj[j] - cj[j] > max)       /* test for optimality */
          {
            optimal = 1;
            max = zj[j] - cj[j];
            enter = j;         /* determine entering variable */
        }
      }
   if (optimal == 1)                        /* not optimal     */
     {

                        /*   the entering variable is xj      */
                        /*   determine the leaving variable   */
      min = 100000000;
      unbounded = 0;
      test1 = 0;

      for (i=1; i<=rows; i++)
      {
           if ((a[i][enter] > 0) && (a[i][b_vector] >= 0))
         {
           test1 = 1;
           unbounded = 1;
           minimum_ratio =  a[i][b_vector] / a[i][enter];
           if (minimum_ratio < min)
           {
              leave = i;
            min = minimum_ratio;
           }
          }
      }
if (test1 == 0) err_message = 1;   /* test for unboundedness */
```

```
        /*   swap the basic variables                                    */
        BASIS[leave].cj_value = cj[enter];
        BASIS[leave].cj_index = enter;

        if (unbounded == 0) optimal = 0;

        /*         pivot                                                 */
        if (optimal == 1)
        {
             /*  divide the entire pivot row by the pivot element       */
             pivot = a[leave][enter];
             for (j = 1; j <= columns; j++)
             {
                  a[leave][j] = a[leave][j] / pivot;
             }

             /*  now update the remaining rows                          */
             for (i = 1; i<= rows; i++)
             {
                if ((i != leave) && (a[i][enter] != 0))
                {
                     row_factor = a[i][enter] / a[leave][enter];
                     for (j = 1; j<= columns; j++)
                {
                a[i][j] = -1* row_factor * a[leave][j] + a[i][j];
             }
          }
        }

    }     /*   optimal - the pivot routine                              */
  }       /*    if (optimal = 1)                                        */
}         /*     end  while  loop                                       */

        if (optimal == 1)    /*  test for non feasibility               */
        {
             for (i = 1; i<= 48; i++)
             {
                if (BASIS[i].cj_index > 84) err_message = 2;
             }
        }

        if (err_message != 0)                /* message(err_message); */

        clrscr();
        exx = 5;
        why = 2;

        /*   the print routine                                           */
        if (err_message == 0)
        {
              clrscr();
              total_samples = 0;
              for (i = 1; i <= 36; i++)
              {
```

```
                    for (j = 1; j<= columns; j++)
                    {
                       b_var = 1;
                       if (BASIS[j].cj_index == i)
                       {
                          b_var = 0;
                          samp[i] = a[j][columns]+l_bound;
                          j = columns;
                       }
                    }
                    if (b_var == 1) samp[i] = l_bound;

                    total_samples = total_samples + samp[i];
                    if (exx > 50)
                    {
                       exx = 5;
                       why = why + 1;
                    }
                    gotoxy(exx, why);
                    cprintf(" %d  %3.1f ",i, samp[i]);
                    exx = exx + 20;

                }

             }                          /*  end if err_message == 0  */


      b = getch();
      b = b + 1;

}
```

Appendix B

C code used for solving the Quadratic Model

```
/*                    MODEL 2                                          */
/*                                                                     */
/* this program makes use of Lemke's algorithm and solves              */
/* the following system  x -Mz -1zo = q                                */
/*                                                                     */

#include <conio.h>
#include <alloc.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

void message(int);

struct basis_info
{
      float cj_value;
      int ys;
 } BASIS[50];

void main()
{

/*                                                                     */
/*   note a[i][98] is the RHS values (Bi) and                          */
/*        a[0][j] is not used                                          */
/*                                                                     */

      float a[50][101];
      float p[40];
      float int_part[40];
      float rhs[13];
      float min;
      float row_factor;
      float minimum_ratio;
      float total_samples;
      float pivot;
      int low_bound=611;
      int columns = 100;
      int rows = 49;
      double intpart, fract;

      int b, iterations = 0, exx, why;
      int i, j, optimal, unbounded, index, zo, new_basic;
      int err_message = 0, old_basis, ray_termination;
      int col_spot, row_spot;

      clrscr();
      textbackground(BLUE);
      textcolor(WHITE);
      /* insert the actual DSC proportions here                        */
```

```
/*       inititialize all coefficients to zero                    */
for (i=1; i<=rows; i++)
{
      for (j=1; j<=columns; j++)
      {
            a[i][j] = 0.0;
      }
}
/*       set up I, the unit matrix 49X49                          */
for (i=1; i<=rows; i++)
{
      for (j=1; j<=rows; j++)
      {
            if (i == j) a[i][j] = 1.0;
      }
}

/*  set up -A matrix.  Since Lemke's uses -M, leave positive      */
/*    the column constraints                                      */
for  (i=1; i<=6; i++)
{
      for (j=63; j<=68; j++)
      {
            a[i][(i-1)*6+j] = -1.0;
      }
}
/*  the row constraints                                           */
for  (i=7; i<=12; i++)
{
      for (j=63; j<=68; j++)
      {
            a[i][6*j-322+i] = -1.0;
      }
}
/*  set up upper bound constraint;  all xij <= low_bound          */
for (j=63; j<=98; j++) a[13][j] = 1.0;

/*  set up A transpose which is negative due to -M                */
for (i = 1; i<= 36; i++)
{
      for (j = 1; j<= 13; j++)
      {
            a[i+13][j+49] = -a[j][i+62];
      }
}
```

```
/*set up H, the negative Hessian matrix of the objective value*/
/*  In this case,   H = -2I                                    */
for (i=1; i<= 36 ; i++)
{
      for (j=1; j <=36; j++)
      {
        if (i == j) a[i+13][j+62] = -2.0;
      }
}
/*  set up column of -1s representing artificial variable Zo  */
for (i=1; i<=rows; i++)
{
      a[i][columns-1] = -1.0;
}
/*   insert column and row RHS values  here                    */
/*  set-up remaining RHS cloumn, which is made up of c         */
for (i=1; i<=36; i++)
{
      a[i+13][columns] = -2*p[i]*low_bound;
}

optimal = 1;
clrscr();
gotoxy(10,5);
cputs("ITERATION # ");

/*  initialization step                                        */
/*   set up basis as the first 48 variables                    */
for (i=1; i<=rows; i++)
{
      BASIS[i].ys = i;
}
min = 100000;
/*  set the artificial variable Zo as basic                    */
for (i=1; i<=rows; i++)
{
      if (a[i][columns] < min)
      {
            min = a[i][columns];
            row_spot = i;
      }
}
col_spot = BASIS[row_spot].ys + rows;
BASIS[row_spot].ys = columns-1;
/*  pivot and update                                           */
for (j=1; j<=columns; j++) a[row_spot][j] = -a[row_spot][j];

for (i=1; i<=rows; i++)
{
      for (j=1; j<=columns; j++)
      {
        if(i != row_spot) a[i][j] = a[row_spot][j] + a[i][j];
      }
}
```

```
/* Lemke's algorithm                                                    */
/* stop when zo leaves basis                                            */
while ((optimal == 1) && (err_message == 0))
{
  gotoxy(30,5);
  cprintf(" %d ",iterations);
      iterations = iterations + 1;
       min = 100000;
       optimal = 0;
       ray_termination = 0;
            for (i=1; i<=rows; i++)
            {
               if (a[i][col_spot] > 0)
               {
                 ray_termination = 1;
                 optimal = 1;
                 minimum_ratio =  a[i][columns] / a[i][col_spot];
                 if (minimum_ratio < min)
                 {
                   row_spot = i;
                   min = minimum_ratio;
                  }
               }
            }

        /*         pivot                                                */
       if (optimal == 1)
       {
        /*  divide the entire pivot row by the pivot element            */
            pivot = a[row_spot][col_spot];
            for (j = 1; j <= columns; j++)
            {
                a[row_spot][j] = a[row_spot][j] / pivot;
            }

            /*  now update the remaining rows                           */
            for (i = 1; i<= rows; i++)
            {
              if (i != row_spot)
              {
                 row_factor = a[i][col_spot] / a[row_spot][col_spot];
                 for (j = 1; j<= columns; j++)
                 {
                   a[i][j] = -1* row_factor * a[row_spot][j] + a[i][j];
                 }
              }
            }

            old_basis = BASIS[row_spot].ys;
            BASIS[row_spot].ys = col_spot;
            if (old_basis > rows) col_spot = old_basis-rows;
            else col_spot = old_basis+rows;
       }
```

```
        /*   optimal - the pivot routine                                    */

         if (old_basis == columns-1) optimal = 0;
        }
        /*     end  while  loop                                             */

        if (err_message != 0) message(err_message);

        clrscr();
        exx = 5;
        why = 2;
        /*   the print routine    */
        if (err_message == 0)
        {

           total_samples = 0;
           for (i = 1; i<= 36; i++)
           {
               for (j = 1; j <= rows; j++)
               {
                 if (BASIS[j].ys == i+62)
                   {

                   fract = modf(a[j][columns], &intpart);
                   if (fract > 0) intpart = intpart + 1;
                   int_part[i] = intpart;
                   total_samples = total_samples + intpart;
                   if (exx > 50)
                   {
                     exx = 5;
                     why = why + 1;
                   }
                   gotoxy(exx, why);
                   cprintf("  %d  %f ",BASIS[j].ys-62, intpart);
                   exx = exx + 20;
                   j = rows;
                 }
               }
        }

        }                                          /* end the print routine  */

        /*  end if err_message == 0                                         */
        b = getch();
        b = b + 1;
        }
```

Appendix C

C code used for solving the Maximum Deviation Model

```
/*  the third model is solved using the simplex algorithm          */
/*   x- and x+ = -y and v respectively                             */

#include <conio.h>
#include <alloc.h>
#include <stdio.h>
#include <math.h>

struct basis_info
{
      float cj_value;
      int cj_index;
} BASIS[86];


void main()
{

      float a[86][161];
      float cj[161];
      float zj[161];
      float samp[86];
      float p[86];
      float ap1[86];
      float int_part[86];
      float pivot = 0;
      float max = 0;
      float min = 0;
      float row_factor = 0;
      float minimum_ratio = 0;
      float z_calc = 0;
      float total_samples = 0;
      float l_bound = 611;
      float row_t=0, col_t =0;
      short rows=85;
      short columns=160;
      double intpart, fract;
      short enter, leave, b, iterations, exx, why;
      short i, j, optimal, b_vector, unbounded;
      short err_message, b_var, test1;

      clrscr();
      textbackground(BLUE);
      textcolor(WHITE);
      b_vector = columns;
      iterations = 0;
      err_message = 0;
      /* insert the actual DSC proportions here                     */
```

```
    /*       inititialize all coefficients to zero                    */
    for (i=1; i<=rows; i++)
    {
          int_part[i] = 0;
          samp[i] = 0;
          for (j=1; j<=columns; j++)
          {
                a[i][j] = 0.0;
          }
    }

    /*  insert the RHS for first 12 rows here                         */


    /*       set column coefficients to equal 1                       */
    /*       for both -y and v                                        */
    for (i=1; i<=6; i++)
    {
          for (j=1; j<=6; j++)
          {
                a[i][37+i+(j-1)*6] = -1.0;
                a[i][73+i+(j-1)*6] = 1.0;
          }
    }

    /*        set row coefficients to equal 1                         */
    /*       for both -y  and v                                       */
    for  (i=7; i<=12; i++)
    {
          for (j=1; j<=6; j++)
          {
                a[i][(i-7)*6+j+37] = -1.0;
                a[i][(i-7)*6+j+73] = 1.0;
          }
    }

 /*  set the surplus variables                                        */
for (i=1; i<=12; i++)
 {
    a[i][i+109] = -1.0;
}

/*  set the lower bound constraint                                   */
for (j = 1; j<= 36; j++)
{
    a[13][j+37] = -1.0;                                  /* -y  */
    a[13][j+73] = 1.0;                                   /*  v  */
}
    /* set up bound artificial variable and RHS                     */
    a[13][158] = 1.0;
    a[13][b_vector] = 0;
```

```
/*          set up x + y - v = pl                                    */
for  (i=14; i<=49; i++)
{
      a[i][i-13] = 1.0;                                    /* x  */
      a[i][i+24] = 1.0;                                   /* -y  */
      a[i][i+60] = -1.0;                                  /* +v  */
      a[i][b_vector] = p[i-13]*l_bound;          /* PL the RHS */
}
/*   calculate     APL                                            */
/*     aPL                                                        */
for (i=1; i<=12; i++)
{
      apl[i] = 0;
      for (j=1; j<=36; j++)
      {
       apl[i] =  apl[i] + (a[i][j+73] * a[j+13][b_vector]);
      }
}

/*    b - APL                                                     */
for (i=1; i<=12; i++)
{
      a[i][b_vector] = a[i][b_vector] - apl[i];
}

/* find the largest RHS of first 12 constraints                   */
max = 0;
enter = 0;
for (i=1; i<=12; i++)
{
  if (a[i][b_vector] > max)
  {
        enter = i;
        max = a[i][b_vector];
  }
}
/* convert all + RHS constraints, except largest RHS, to - RHS   */
  for (i=1; i<=12; i++)
  {
    if (a[i][b_vector] > 0)
    {
        if (i != enter)
        {
          a[i][b_vector] = a[i][b_vector] - a[enter][b_vector];
          for (j=38; j<=121; j++)          /* used to be to 121  */
          {
              a[i][j] = a[i][j] - a[enter][j];
          }
        }
    }
  }
```

```
/* convert all -RHS to +, and convert -I to +I for the basis  */
for (i=1; i<=12; i++)
{
      if (a[i][b_vector] <= 0)
      {
            for (j=38; j<=columns; j++)
            {
                  a[i][j] = -a[i][j];
            }
      }
}

/*  need 1 artificial variable                                 */
 if (enter > 0) a[enter][159] = 1;

/*  set up z vector   -z + y + v <= 0    multiply by -1        */
for (i=50; i<=85; i++)
{
      a[i][37]  = -1.0;                                  /*  -z  */
      a[i][i-12] = 1.0;                                  /*  y   */
      a[i][i+24] = 1.0;                                  /*  v   */
      a[i][i+72] = 1.0;                /* the slack variables */
}

/*  the objective function coefficients                        */
for (i=1; i<=columns-1; i++) cj[i] = 0;
                                                     /*  z = 1  */
cj[37] = 1.0;

/*  set the 2 big M variable's coefficients                    */
for (i=158; i<=159; i++) cj[i] = 100000,
i=0;
/* set the 11 surplus variables and 1 artificial as basic      */
for (j=110; j<=121; j++)
{
      i++;
      if ((j - 109) != enter)
      {
            BASIS[i].cj_index = j;
            BASIS[i].cj_value = cj[j];
      }
      else
      {
            BASIS[i].cj_index = 159;
            BASIS[i].cj_value = cj[159];
      }
}

/*  set 1 artificial variable as basic var                     */
for (j=158; j<=158; j++)
{
      BASIS[13].cj_index = j;
      BASIS[13].cj_value = cj[j];
}
```

```
/*  set 36 x variables as basic                                    */
for (j=1; j<=36; j++)
{
      BASIS[j+13].cj_index = j;
      BASIS[j+13].cj_value = cj[j];
}

/*  set 36 slack variables as basic                                */
for (j=122; j<=157; j++)
{
      BASIS[j-72].cj_index = j;
      BASIS[j-72].cj_value = cj[j];
}
optimal = 1;
clrscr();
gotoxy(10,5);
cputs("ITERATIONS # ");

/* the simplex algorithm                                           */
while ((optimal == 1) && (err_message == 0))
{
       gotoxy(30,5);
       cprintf(" %d ",iterations);
       iterations = iterations + 1;

      max = 0;
      optimal = 0;
      for (j=1; j<=columns-1; j++)
      {
        z_calc = 0;
        for (i=1; i<=rows; i++)   /*  calculate Zjs                */
        {
           {
            z_calc = z_calc + a[i][j]*BASIS[i].cj_value;
           }
         }
         zj[j] =  z_calc;
         if (zj[j] - cj[j] > max) /* test for optimality           */
         {
           optimal = 1;
           max = zj[j] - cj[j];
           enter = j;    /* determine entering variable           */
         }
      }
```

```
if (optimal == 1)                                    /* not optimal */
{

  /*  the entering variable is xj                                    */
  /*  determine the leaving variable                                 */

      min = 1000000;
      unbounded = 0;
      test1 = 0;
      for (i=1; i<=rows; i++)
      {
        if ((a[i][enter] > 0) && (a[i][b_vector] >= 0))
        {
          test1 = 1;
          unbounded = 1;
          minimum_ratio =  a[i][b_vector] / a[i][enter];
          if (minimum_ratio < min)
          {
             leave = i;
              min = minimum_ratio;
           }
         }
      }
if (test1 == 0) err_message = 1;    /* test for unboundedness */
/*   swap the basic variables                                       */
BASIS[leave].cj_value = cj[enter];
BASIS[leave].cj_index = enter;

if (unbounded == 0) optimal = 0;
/*   pivot                                                          */
if (optimal == 1)
{

  /*  divide the entire pivot row by the pivot element              */
  pivot = a[leave][enter];
  for (j = 1; j <= columns; j++)
  {
    a[leave][j] = a[leave][j] / pivot;
   }

   /*  now update the remaining rows                                */
   for (i = 1; i<= rows; i++)
   {
      if ((i != leave) && (a[i][enter] != 0))
      {
        row_factor = a[i][enter] / a[leave][enter];
        for (j = 1; j<= columns; j++)
        {
          a[i][j] = -1* row_factor * a[leave][j] + a[i][j];
        }
     }
   }
}                          /*   optimal - the pivot routine    */
```

```
        }                                   /*    if (optimal = 1)      */

    }                                       /*    end  while  loop      */


if (optimal == 1)                          /*  test for non feasibility */
{
  for (i = 1; i<= rows; i++)
  {
    if (BASIS[i].cj_index > 157) err_message = 2;
   }
}

if (err_message != 0)   /* message(err_message); */

clrscr();
exx = 5;
why = 2;


/*   the print routine                                                    */
if (err_message == 0)
{
     clrscr();
       total_samples = 0;
       for (i = 1; i<= 36; i++)
       {
         max = 0;
         for (j = 1; j <= rows; j++)
         {
             if (BASIS[j].cj_index == i)
             {
                max = 1;
                intpart = 0;
                fract = modf(a[j][columns], &intpart);
                if (fract > 0) intpart = intpart + 1;
                int_part[i] = intpart;
                total_samples = total_samples + intpart;
                if (exx > 60)
                {
                   exx = 5;
                   why = why + 1;
                }
                gotoxy(exx, why);
                cprintf("  %4.1f ", intpart);
                exx = exx + 10;
                j = rows;
             }
         }
```

```
            if (max == 0)
            {
                if (exx > 60)
            {
             exx = 5;
             why = why + 1;
        }
        gotoxy(exx, why);
        cprintf("  %4.1f ", 0.0);
        exx = exx + 10;

       }
      }

}                                                    /* end the print routine  */
gotoxy(5, why + 3);

b = getch();
b = b + 1;

}
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704 0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 1992	Final

4. TITLE AND SUBTITLE
Optimal Sampling Plans for Items Representing Two Population Groups

5. FUNDING NUMBERS

6. AUTHOR(S)
Randal S. Wendell

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
HQ Defense Logistics Agency
Operations Research & Economic Analysis Office (DLA-LO)
Cameron Station
Alexandria, VA 22304-6100

8. PERFORMING ORGANIZATION REPORT NUMBER
DLA-92-P20041

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
HQ Defense Logistics Agency
Laboratory Testing Team (DLA-QL)
Cameron Station
Alexandria, VA 22304-6100

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Public Release; Unlimited Distribution

12b. DISTRIBUTION CODE

13. ABSTRACT
This paper examines a problem of determining optimal sampling plans. The items to be sampled belong to two distinct populations which are partitioned into sub-populations that require sampling plans. Three mathematical programming models are investigated that minimize the total sample size while also ensuring that the proportion of samples closely resemble the actual population proportions. Both linear and non-linear programming techniques are used to find an optimal sampling plan. Finally, comparisons are made from the solutions generated by "real" data for all three models.

14. SUBJECT TERMS
Sampling, Laboratories, Testing

15. NUMBER OF PAGES
83

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	